L-Modified ILP Evaluation Functions for positive-only Biological Grammar Learning

Thierry Mamer[1] ⋆, Christopher H Bryant[2], and John McCall[1]

[1] School of Computing, The Robert Gordon University,
St. Andrews Street, AB25 1HG, Aberdeen, Scotland, UK.
http://www.comp.rgu.ac.uk
[2] School of Computing, Science and Engineering, Newton Building,
University of Salford, Salford, Greater Manchester, M5 4WT, UK.

Abstract. We identify a shortcoming of a standard positive-only clause evaluation function within the context of learning biological grammars. To overcome this shortcoming we propose L-modification, a modification to this evaluation function such that the lengths of individual examples are considered. We use a set of bio-sequences known as neuropeptide precursor middles (NPP-middles). Using L-modification to learn from these NPP-middles results in induced grammars that have a better performance than that achieved when using the standard positive-only clause evaluation function. We also show that L-modification improves the performance of induced grammars when learning on short, medium or long NPPs-middles. A potential disadvantage of L-modification is discussed. Finally, we show that, as the limit on the search space size increases, the greater is the increase in predictive performance arising from L-modification.

Key words: Inductive Logic Programming (ILP), Biological Grammar Induction, Machine Learning, Minimum Description Length (MDL)

1 Introduction

This work aims to improve the automated learning of biological grammars using Inductive Logic Programming (ILP) tools.

1.1 Biological grammars

Biological grammars (BG) are patterns in the form of grammars that model biological sequences: among others, protein sequences. Linguistic approaches can be used for representing the structure of proteins [11] because their primary structure can be represented as a sequence of characters from a well defined chemical alphabet of only 20 different amino-acids (A, C, D, E, F, G, H, I, K, L, M, N,

⋆ Corresponding Author: Thierry Mamer, School of Computing, The Robert Gordon University, St. Andrews Street, AB25 1HG, Aberdeen, Scotland, United Kingdom; E-mail: tm@comp.rgu.ac.uk; webpage: http://www.comp.rgu.ac.uk/staff/tm/

P, Q, R, S, T, V, W, Y). These sequences can be of any length, from very small, up to hundreds of characters long. Formal grammars can define dependencies in biological sequences because of their declarative and hierarchical nature (e.g. biological sequences folding up in three dimensional space lead to dependencies between distant parts). Using grammars to model biological sequences brings two main advantages to the biologist: first, grammars can be used to annotate sequences whose function is yet unknown and thus suggest a likely function; second, because the grammar structure represents common points between sequences of similar functions, they could help biologists to understand biological functions. See Figure 1 for a basic example of a BG parsing a protein sequence. BGs can take several forms depending on the approach taken. In our experiments, BGs take the form of *context free grammars* (CFG) (see Section 2.2) and describe a specific family of proteins (see Section 2.1).

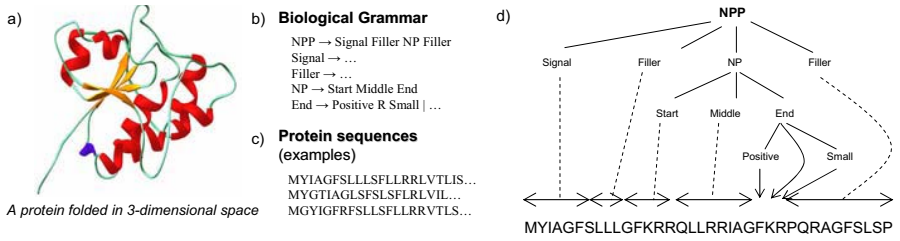

Fig. 1. Figure 1a) shows an example of how a protein might look like in 3 dimensional space. Figure 1c) shows a few examples of protein sequences. These are sequences of the amino acids which make up the proteins, but ignoring the 3 dimensional structure. Figure 1b) gives an indication of what a biological grammar describing an Neuropeptide Precursor Protein (NPP) might look like; a set of production rules that can parse a protein sequence. Figure 1d) shows a very basic example of a parse tree, illustrating how an NPP sequence could be parsed by an NPP grammar.

In our experiments we learn BGs from examples of protein sequences belonging to a certain family of proteins (see Section 2.1). Protein sequences generally have highly variable length, which is what sets our task of learning BG apart from most traditional ILP learning tasks. It is this variable length of examples in the training data given to the ILP system that is the main focus of this work.

1.2 Biological Grammar Learning with ILP

Muggelton et al [8] first investigated Chomski-like grammar representations for learning cost-effective, comprehensible predictors of members of biological sequence families. They used the ILP tool CProgol [6] to learn grammars describing neuropeptide precursors (NPPs). Their best predictor made the search for novel NPPs more than one hundred times more efficient than randomly selecting proteins for synthesis and testing them for biological activity. Our work takes

its roots in the approach of [8] as we use *definite clause grammars* DCG [9] (see Section 2.2) and we also use a subset of the NPP dataset used in [8](see Section 2.1). Bryant et al. [1, 3, 2], which also takes its roots in the approach of [8] used the ILP tool Aleph [12] because Aleph is a modular system, it is easier to modify and gives a large number of options. Our work also uses Aleph as we also need the ability to customize the clause evaluation functions and the clause coverage computation (see Section 3.4).

1.3 Hypothesis

The hypothesis that we want to test in this work is as follows:

When using the generic ILP system Aleph to learn biological grammars describing proteins of the NPP family, then the predictive performance of these grammars can be improved by estimating the quality of a learned clause through an evaluation function that takes into account the length of the examples in the training data.

1.4 Justification

To decide between two models that equally well describe the data, the minimum description length principle (MDL) [10] suggests that $I(M|E)$ (which is the description length I of the model M, given the examples E) should be minimized. In our experiments we consider $I(M|E)$ to be the sum of the description lengths of the model $I(M)$ and the examples encoded through the model $I(E|M)$. An approximation to this principle is to estimate the compression that would be achieved by encoding the examples with the model [13, 5]. To estimate this compression, the size of the examples that would be encoded by the model has to be known. Under the assumption that each example has the same length, which is usually the case, their individual length can be neglected and a simple count of covered examples can be used to estimate the compression achieved. However if we are learning BGs, the examples given are biological sequences, which often have variable length. In that case the length of each individual example has an undeniable impact on the total size of examples encoded through the model. It then becomes clear that the length of individual examples should not be neglected while estimating the compression achieved by a model. We want to apply this idea to an evaluation function that is to evaluate a grammar clause during learning. Such an evaluation function would use the size of the clause and the size of all examples that are covered by that clause to evaluate the performance of the clause. To the knowledge of the authors there has been no previous work on positive-only clause evaluation functions, used in ILP, that consider the length of training examples. Evaluation functions that estimate the compression achieved by a model are often called compression measures.

2 Experiment Design

In our experiments we run the ILP tool Aleph [12] on the NPP datasets described in Section 2.1. For each of those datasets we use several different clause evaluation functions (as described in Section 3) and observe the differences in performance. The results of the training and testing are recorded and subsequently summarized in Section 4.

2.1 Data Sets

Dataset 1 - Whole set
For the first set of experiments we used a part of the human Neuropeptide Precurser Protein (NPP) dataset which was used in [8]. The experiments conducted by [8] involved, among others, the inference of BGs on a set of NPP sub-sequences called *middle* (henceforth denoted as *NPP-middles*). In this work, we will only consider the NPP-middles because these are most interesting in relation to our hypothesis (Section 1.3). First, the NPP-middles are considerably longer than any of the other NPP parts which makes the induction of a grammar describing them a more challenging task, and second, they are also the only NPP subsequences that display high variations in length; our NPP-middles range from 5 to 95 amino acids. This dataset consists of 76 positive examples, 2908 random examples and some Background Knowledge (BK).

Dataset 2 - Training data grouped by length
To see if the evaluation functions perform differently for longer or shorter examples we decided to split up the dataset into several parts and conduct a second set of experiments. We took the dataset discussed in the previous paragraph and split it into three disjoint subsets, based on the length of positive examples, effectively creating three separate datasets. We denote these three subsets as follows: NPP-middles-short, NPP-middles-medium and NPP-middles-long. The intention was to split the set of positive examples into three sets containing more or less the same number of examples. The first subset, NPP-middles-short, contains 24 examples each of length (number of characters) $l < 13$. The reason why this set contains only 24 examples instead of 25 or 26, as seems logical with 76 total positive examples, is because we set a length threshold of 13 in order to prevent the examples of length 13 being split between two subsets. The second subset, NPP-middles-medium contains 26 examples with $13 \leq l \leq 29$. The third subset, NPP-long contains 26 examples with $l > 29$. The random examples were split up in the same way according to length of their examples, using the same cut-off values that were used for the positive examples. This results in NPP-middles-short containing 908 random examples, NPP-middles-medium containing 864 random examples and NPP-middles-long containing 1136 random examples.

Background Knowledge (BK)
The BK in this dataset consists of general molecular biology knowledge which can be considered relevant for any protein grammar inference process. The BK contains amino acid letters and their physio-chemical properties (as first proposed by [4], and also used by [8]) and gaps. The purpose of gaps is to match parts of the protein sequence that are not directly relevant to the function or which cannot be characterized by the provided background predicates, but which still participate in the overall structure of the molecule [2]. This dataset, including the BK, was also used in [3] and [1]. (The datasets and BK can be found at: www.comp.rgu.ac.uk/staff/tm/materials/ILP08/) The set of random examples may contain protein sequences that would be positive, but at the time this dataset was collected, were still undiscovered as such. Consequently we cannot treat them as negative examples so we have to use a positive-only learning approach [7].

A 5-fold stratified cross-validation was performed on all datasets. The stratification of the cross validation was based on the length of the examples, ensuring that all training and test sets include a variety of examples of different lengths. The same Background Knowledge is used in each experiment.

2.2 Representation of biological grammars and sequences

We are using the ILP tool Aleph to learn biological grammars. All input given to Aleph is using a Prolog related syntax. The same goes for the induced result, especially since we might want to use the induced grammar in further tests or experiments using Aleph or Prolog. The resulting BG, a context free grammar, is a set of rules that represent a given set of protein sequences. To represent such a grammar we use a Definite Clause Grammar (DCG) formalism [9] in the same way as in [8, 1, 3, 2]. DCGs require sequences to be represented by a list, where each element in this list stands for a letter in the sequence. DCG rules take such a list as input and pass it on to the predicates that make up the rule. Each predicate, starting with the first, then matches one or more elements from the start of the list and returns the rest. This new, shorter list is then in turn given to the next predicate in the rule. If the last predicate returns an empty list, then the whole sequence is matched by the grammar rule and we consider the sequence to be covered by that rule. Aleph learns one rule at the time until all the examples are covered, and then it puts all the induced rules together to form the resulting grammar. (See Table 2 on page 8 for a summary of Aleph's search algorithm)

2.3 Suitable performance measure for an induced grammar

In Machine Learning and more specifically ILP, the most popular performance measure used to evaluate the final result of learning is the predictive accuracy:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP stands for true positives, FP for false positives, TN for true negatives and FN for false negatives However the accuracy of an induced grammar might not be a suitable performance measure when learning biological grammars. Our dataset (see Section 2.1) contains random examples instead of negative ones, so instead of TN and FN, we get TR and FR, referring to true and false randoms. These values can still be put in the above formula to compute the accuracy, but they don't quite mean the same thing. Our dataset also has a considerable unbalance in the ratio between positive and random examples: 76 positives compared to 2908 randoms. A consequence of this is that despite covering different numbers of the considerably rare positives, induced grammars have a high chance of being awarded a very high accuracy by excluding most of the abundant random examples [8]. To prevent cases where the accuracy could be inconclusive, we considered other, additional quality measures for the induced grammars.

From the domain of Information Retrieval (IR) we considered precision, recall and F-measure [14]. *Precision* in IR is the fraction of predicted positive examples that are indeed true positives and *Recall* is the fraction of true positives among all positives:

$$precision = \frac{TP}{TP + FP} \quad ; \quad recall = \frac{TP}{TP + FN}$$

In IR these two measures are often used in conjunction with the F-measure, which is the weighted harmonic mean of precision and recall:

$$F - measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

These measures seem appropriate for our domain as well, so we decided to include them in this work (see Table 4 on page 9).

3 Clause Evaluation Functions

The focus of this work is the evaluation function which the ILP system uses to evaluate a clause. Such a clause evaluation function gives each accepted clause a score estimating its quality (Table 2 step 3). After the search is complete, the clause with the best score is added to the grammar (Table 2 step 4). This section describes the different clause evaluation functions used in this work. A summary can be found in Table 1.

Note here that a clause evaluation function and a grammar performance measure (introduced in Section 2.3) are two distinct things. While a clause evaluation function is used to grade each individual clause during learning, a grammar performance measure is only used after learning is completed successfully. The grammar performance measure aims to estimate the future performance of the entire learned grammar, which consists of a number of clauses.

Table 1. List of all evaluation functions used in this work

(1)	Score = $log(P) \quad -log(\frac{R+1}{Rsize+2}) \quad -\frac{L}{P}$
(2)	Score = $log(PosCoverage) \; -log(\frac{RanCoverage+1}{RLsize+2}) \; -\frac{L}{PosCoverage}$
(3)	Score = $log(PosCoverage) \; -log(\frac{RanCoverage+1}{Rsize+2}) \; -\frac{L}{PosCoverage}$
(4)	Score = $log(PosCoverage) \; -log(\frac{RanCoverage+1}{Rsize+2}) \; -L$

3.1 Standard positive-only evaluation function

The standard positive-only learning evaluation score in ILP was devised by Muggelton [7]:

$$Score = log\,(P) - log(\frac{(R+1)}{(Rsize+2)}) - \frac{L}{P} \tag{1}$$

Where P is the number of positives covered; R is the number of randoms covered; $Rsize$ is the total number of randoms and L is the number of literals in the hypothesis. This evaluation function has been widely used by the community and therefore is our benchmark evaluation function. This means that our experiments aim to find a function that outperforms this one.

3.2 L-modification of the standard positive-only evaluation function

Function (1) does not consider the length of individual examples, so to put our hypothesis to the test we had to modify it. What we propose is to replace any variable in evaluation function (1) that would refer to numbers of examples (may that be covered, not covered or total) with a different variable that instead refers to a modified value which takes into account the length of examples. This means more precisely that instead of feeding P and R to to the evaluation function, we replaced all their occurrences in (1) with *PosCoverage* and *RanCoverage* respectively. PosCoverage is the sum of the lengths of all covered positive examples and RanCoverage is the sum of the lengths of all covered random examples. These changes combine the idea of existing pos-only evaluation functions with the idea of considering the length of examples. Henceforth we denote such a change of variables as *L-modification*. In addition to P and R, the variable $Rsize$ in (1) also refers to a number of examples: the total number of random examples in the training data. As the name of the variable suggests, this number intends to represent the size of the set of randoms, so within the context of this work, it makes sense that we also apply our L-modification to this variable. This means that we replace $Rsize$ with $RLsize$, which is the sum of the lengths of all random examples in the training data. Applying these modifications gives us:

$$Score = log(PosCoverage) - log(\frac{RanCoverage+1}{RLsize+2}) - \frac{L}{PosCoverage} \tag{2}$$

3.3 Further L-modified evaluation functions used in this work

Although our experiments will focus on evaluation functions (1) and 2 we also ran our experiments on two other modified versions of evaluation function (1). The first of these only replaces P and R with *PosCoverage* and *RanCoverage* respectively with no further changes:

$$Score = log(PosCoverage) - log(\frac{RanCoverage + 1}{Rsize + 2}) - \frac{L}{PosCoverage} \quad (3)$$

The second one also takes into account that the value of PosCoverage is 10 to 60 times larger than P, therefore the last term of function 3, $(\frac{L}{PosCoverage})$, possibly leads to the clause length L greatly losing influence on the score. Therefore we tried giving L more weight by not dividing by $PosCoverage$:

$$Score \ = \ log(PosCoverage) - log(\frac{RanCoverage + 1}{Rsize + 2}) - L \quad (4)$$

3.4 Implementing L-modifications

Three of the four evaluation functions in this work use the L-modified coverage instead of the traditional coverage. Aleph, the ILP tool we are using does not have features that can access the length of individual examples in the training data. However it allows for user defined clause evaluation functions. Therefore, instead of having Aleph calculate the coverage of a clause by itself, we had it use customized predicates to compute the coverage during clause evaluation. These predicates parse examples through the clause and return the modified coverage. They still respect the search algorithm that Aleph applies (see Table 2), i.e. they only parse those examples that are still uncovered (not yet deemed redundant). In our experiments, when using the clause evaluation function (1) our predicates calculate the number of positive and random examples covered, just like Aleph's functions would, however for all subsequent experiments, using functions (3), (4) and (2) our predicates calculate the L-modified coverage, e.g. the sum of the lengths of all the covered positive or random examples. Some of the predicates that were used in this work to compute the L-modified coverage can be found in Appendix B on page 16.

Table 2. A simplification of the basic Aleph algorithm

1. Select a positive example to be generalised. If none exist, stop.
2. Build the bottom clause; the most specific clause entailing the example selected.
3. Search; Find a clause more general than the bottom clause. (Construct a search tree, each node containing a clause which consists of a subset of the literals in the bottom clause. Search for the clause with the best score)
4. Remove redundant. The clause with the best score is added to the grammar. All examples made redundant are removed. Return to Step 1.

Table 3. Summary of the results on all datasets (see Section 2.1) - The first leftmost column indicates which evaluation function was used (see Section 3 or Table 1), all the subsequent columns give the sum, average and standard deviation (std) of true positives (TP), false positives (FP), true randoms (TR) and false randoms (FR) observed during testing.

Experiments conducted on NPP-middles												
evalfunc	TP			FP			TR			FR		
	sum	av.	std	sum	av.	std	sum	av.	std	sum	av.	std
(1)	50	10	2.12	652	130.4	72.44	2256	451.2	72.28	26	5.2	2.39
(2)	39	7.8	2.39	29	5.8	1.92	2879	575.8	1.79	37	7.4	2.61
Experiments conducted on NPP-middles-short												
(1)	20	4	1.00	62	12.4	9.13	846	169.2	8.93	4	0.8	1.10
(2)	15	3	0.71	5	1	1.00	903	180.6	1.34	9	1.8	1.10
Experiments conducted on NPP-middles-medium												
(1)	13	2.6	2.30	76	15.2	10.83	788	157.6	10.83	13	2.6	2.61
(2)	14	2.8	1.64	9	1.8	1.92	855	171	2.35	12	2.4	1.82
Experiments conducted on NPP-middles-long												
(1)	6	1.2	0.84	170	34	26.45	966	193.2	26.48	20	4	0.71
(2)	2	0.4	0.55	19	3.8	2.68	1117	223.4	2.88	24	4.8	0.8

Table 4. Evaluation of the results, derived from the values given in Table 3 - The first leftmost column indicates which dataset the values are referring to, the following column states which evaluation function (evalfunc) was used (see Section 3 or Table 1) and the next columns give the average of accuracy, precision, recall and F-measure observed during testing

Dataset	evalfunc	av. Accuracy	av. Precision	av. Recall	F-measure
NPP-middles	(1)	0.77	0.08	0.66	0.14
	(2)	0.98	0.57	0.52	0.54
NPP-middles-short	(1)	0.93	0.24	0.83	0.38
	(2)	0.98	0.75	0.62	0.68
NPP-middles-medium	(1)	0.90	0.15	0.50	0.23
	(2)	0.98	0.61	0.54	0.57
NPP-middles-long	(1)	0.84	0.03	0.23	0.06
	(2)	0.96	0.10	0.08	0.09

4 Results

Table 3 shows the results obtained from running evaluation functions *(1)* and (2) on all datasets. From the results of the 5-fold cross validation we get the sum, average and standard deviation of TP (true positives), FP (false positives), TR (true randoms) and FR (false randoms).

Table 4 shows the evaluation of all the data collected in Table 3; the accuracy, precision, recall and F-measure of a theory. Definitions of these performance measures, can be found in Section 2.3.

5 Discussion

5.1 Effects of the L-modifications

Dataset NPP-middles
We use the experiment using clause evaluation function (1) as our benchmark and compare the results of the other experiments with this one. The main change that we can observe when looking at Table 3 is that the FP rate has been decreased drastically by the L-modified experiments. FP of 5.8 is an acceptable value, even within the positive-only learning NPP domain. It is mainly a consequence of this change in FP that the accuracy was increased from 0.77 to 0.98.

However, as we suggested in Section 2.3 this high accuracy could be misleading. The precision has been increased from 0.08 to 0.57 which is a considerable change. (1) has an extremely low precision as on average 130 random examples are accepted by the theory learned. This is over 25% of the total randoms provided in each fold.

The recall has been slightly decreased from 0.66 to 0.52. The reason for this is that the L-modified experiments produce theories with lower TP: 7.8 as opposed to 10 by our benchmark experiment.

Finally, the F-measure, the weighted harmonic mean of precision and recall, is increased in the L-modified experiments: from 0.14 to 0.54. We see this as a significant improvement.

Dataset NPP-middles-(short,medium,long)
Concerning all 3 subsets of this dataset, the same observations can be made as in the previous paragraph, using NPP-middles: the accuracy increases as a consequence of FP decreasing, the precision improves as well, the recall decreases, except using NPP-middles-medium where is increased by only 0.04 and the F-measure finally increases as well.

However a few additional observations can be made here. Looking at Figure 2 we can see that for each subset of the NPP-middles (short, medium and long) the performance of the L-modified evaluation functions is higher than that of our benchmark function (1). Also, generally, for each measure, the performance is better for shorter examples than for longer ones. The reason for this is that it

is easier to learn rules covering shorter examples than longer ones, which makes sense.

Another observation that can be made when looking at Table 4 is that for datasets NPP-middle-short and NPP-middle-medium the accuracy and F-measure are higher than for dataset NPP-middles, even without L-modification. This is quite interesting as one would not expect this to be the case. Clearly, datasets NPP-middle-short and NPP-middle-medium, being subsets of NPP-middles, contain less examples than NPP-middles, so one would expect the performance to be lower. What sets the smaller datasets apart from the larger one is that the variation in the length of examples is different. This seems to indicate that the greater variations in the length of the examples contained in the NPP-middles dataset, compared to that in each of its subsets, make it harder for Aleph to generate hypotheses with similar performance.

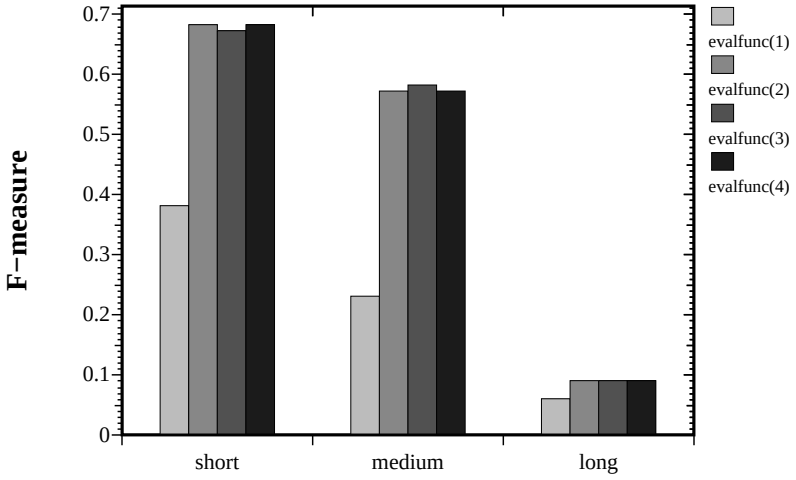

Fig. 2. for all three parts of dataset NPP-middles-(short, medium and long) the performance in terms of F-measure of each of the four evaluation functions is plotted

5.2 Comparing clause evaluation functions (3), (4) and (2)

In this work, we limited the reporting of results to evaluation functions (1), which served as our benchmark, and (2), which is the main contribution of this work. However as we stated in Section 3.3 we also ran all experiments reported using evaluation functions (3) and (4). The results of using these functions were not reported in Tables 3 and 4 because they were very similar, and in many cases identical to the outcomes of the experiments using function (2). Furthermore when we learn on the entire NPP-middle dataset, without any cross validation, then the grammars learned using these 3 evaluation functions are in fact identical and the search constructs the same number of nodes for these 3 experiments.

There is a slight variation in time needed which can be accounted for by different computation times of the slightly different evaluation functions. We can conclude that although there is a significant difference in performance when introducing the length of examples into the clause evaluation function, there is no further advantage to be gained by considering the differences between (2), (3) and (4). The most likely explanation for this is that the first term of these L-modified equations, $log(PosCoverage)$, is the dominant term, especially in these experiments where we deal with large values of $PosCoverage$. As a consequence, the second and third terms in the L-modified equations have only a minor influence on the score.

5.3 Analysis of induced grammars

We have already noted that using our L-modified clause evaluation function increases the accuracy and F-measure of the learned grammars. However we made other observations that need mentioning. Table 5 contains an analysis of the grammars that were learned using the NPP-middle dataset, without any cross validation.

Applying L-modification to the evaluation function increases the number of rules in the learned grammars. This is the case because more rules cover only one positive example each (see Table 5 column 4). So even though the grammars are better at describing the positive examples (cover a lot less random examples), they are more complex. This could be an indication that the grammars are more likely to be overfitted to the dataset. It is worth noting that those rules that cover only one example usually cover a very large example. Also, rules that cover only one example could provide useful information to the experts of the domain as they may indicate which part of a protein is of biological significance.

Table 5. Details of the grammars induced using the NPP-middle dataset, without any cross validation - The first column indicates which evaluation function from Section 3 was used while inducing the grammar, the second column gives the average number of rules that make up the induced grammars when learned using 5-fold cross validation, the third column gives the number of rules that make up the induced grammar when learned on the whole dataset and the fourth column gives a count of how many rules were induced that only cover one positive example when learned on the whole dataset

evaluation function	av. # of rules per fold	# of rules in whole dataset	# of rules covering 1 ex.
(1)	15	14	3
(2)	30.6	35	22

5.4 Total time

In our experiments we also recorded the total time needed for the induction process. In general, experiments using functions (2), (4) and (3) took a lot longer

than those using function (1). Even though computing our L-modified coverage requires a little more computational power, it is more likely that the large increase in running time is a consequence of more nodes being constructed by the search, which in turn results in the increase of performance of the resulting grammars.

In order to confirm this we ran an additional set of experiments using the NPP-middles dataset. This time we used a number of different parameters for `setNodes`, the Aleph setting that controls how many nodes are constructed during each search. We used the following values: 100000 (the value we used in all other experiments), 50000, 10000, 5000 and 1000. We then recorded the total number of nodes constructed for each learning task (the sum of all nodes constructed during each search). Figure 3 shows the graph plotting the performance of the evaluation functions against the total number of nodes constructed. We can observe that more nodes are constructed to learn grammars with better performance. In addition to that, the rate of increase is greater for the L-modified evaluation functions than for the benchmark function. This shows that indeed, for larger search spaces, the L-modified evaluation functions result in grammars with a higher F-measure. There are two factors that are responsible for the search space being larger when using L-modified evaluation functions:

1. If L-modified clause evaluation functions are used to calculate the score for a clause, the search is not as easily satisfied and more iterations of the search algorithm are called before a clause is finally added to the resulting grammar.
2. As we have shown in Section 5.3, grammars learned using the L-modified scores consist of more rules than their unmodified counterpart, which means that more examples are chosen to be generalized by step 1 of the search algorithm (see Table 2).

6 Conclusions

We have shown that when learning on the NPP-middle dataset, the L-modifications we propose do improve the performance of the induced grammars, both in terms of accuracy and F-measure. Splitting the NPP-middle dataset in 3 disjoint subsets and learning on those, we have shown that our L-modification improves performance of induced grammars for short, medium and long examples. Within this context, we have also shown that it is generally harder to learn rules covering longer examples than shorter ones. By observing the outcomes of learning on NPP-middle dataset and comparing them with those that we can observe when learning on each of its subsets (NPP-middle-short, -medium and -large) we conclude that it is harder for the ILP tool used in this work (Aleph) to learn from examples that have a high variation in their lengths. Finally, by running several experiments setting different limits for the amount of nodes allowed to be constructed during learning, we have shown that for larger search spaces, the L-modified evaluation functions result in grammars with a higher F-measure and that the rate of improvement is higher using L-modified functions.

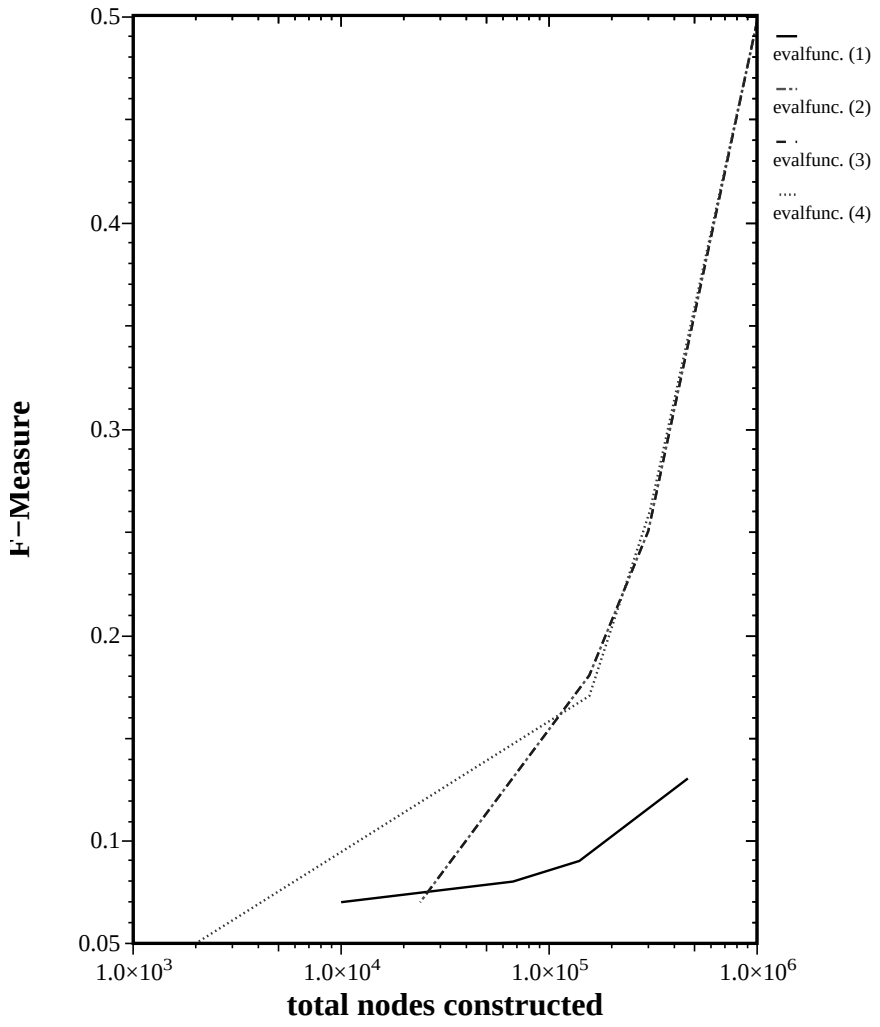

Fig. 3. for each evaluation function (evalfunc) the performance is plotted against the total number of nodes constructed at the end of induction

Considering all the above, we can conclude that the L-modification proposed in this work does indeed improve the performance of evaluation function (1). Therefore we would expect the L-modification to improve other clause evaluation functions as well, so now there is a need to apply this approach to more than one standard clause evaluation function.

7 Future Work

Applying L-modification to different evaluation functions

To fully support our claims that the L-modification proposed in this work improves the grammars that are learned, there is a need to apply this approach to other standard clause evaluation functions. However, this is complicated by the fact that so far, we were dealing with positive-only learning. Evaluation functions meant for positive and negative learning are more common. We therefore propose to investigate how to apply this approach of L-modifying the coverage computation to clause evaluation functions which are tailored to positive and negative learning.

References

1. C.H. Bryant and D. Fredouille. A parser for the efficient induction of biological grammars. In S. Kramer and B. Pfahringer, editors, 15^{th} *International Conference on Inductive Logic Programming: late-breaking paper track.*, pages 3–8. University of Bonn, Bonn, Germany, July 2005. http://wwwbib.informatik.tu-muenchen.de/infberichte/2005/TUM-I0510.idx.
2. C.H. Bryant, D. Fredouille, A. Wilson, C.K. Jayawickreme, S. Jupe, and S. Topp. Pertinent background knowledge for learning protein grammars. In J. Furnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Proceedings of the 17th European Conference on Machine Learning*, number 4212 in Lecture Notes in Artificial Intelligence, pages 54–65. Springer-Verlag, Berlin, September 2006.
3. D. Fredouille, C. H. Bryant, C. K. Jayawickreme, S. Jupe, , and S. Topp. An ILP refinement operator for biological grammar learning. In S. Muggleton, editor, 16^{th} *International Conference on Inductive Logic Programming*, 2006.
4. S. Muggleton, R.D. King, and M. J. E. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering -Oxford-*, 5(7):647, 1992.
5. S. Muggleton, A. Srinivasan, and M. Bain. Compression, significance and accuracy. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Machine Learning Conference*, pages 338–347. Morgan-Kaufmann, 1992.
6. S. H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
7. S. H. Muggleton. Learning from positive data. In S. H. Muggleton, editor, *Proceedings of the 6th International Workshop on Indictive Logic Programming*, volume 1314, pages 358–376. Springer Verlag, 1996.
8. S. H. Muggleton, C. H. Bryant, A. Srinivasan, A. Whittaker, S. Topp, and C. Rawlings. Are grammatical representations useful for learning from biological sequence data? - a case study. *Journal of Computational Biology*, 8(5):493–522, October 2001.

9. F Pereira and D Warren. Definite clause grammars for language analysis. *Readings in natural language processing*, pages 101–124, 1986.
10. J. J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.
11. D. B. Searls. Linguistic approaches to biological sequences. *Computer Applications in the Biosciences*, 13(4):333–344, 1997.
12. A. Srinivasan. A learning engine for proposing hypotheses (Aleph). http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph, 1993.
13. A. Srinivasan, S. Muggleton, and M. Bain. The justification of logical theories based on data compression. *Machine Intelligence*, 13:91–125, 1994.
14. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.

Appendix A: Materials

Most of the materials used in this work, dataset and input files for Aleph can be found online at www.comp.rgu.ac.uk/staff/tm/materials/ILP08/.

Appendix B: Code for L-modified coverage computation

predicate: compute_Lmodcover/3

This predicate computes the L-modified coverage of a clause on all remaining examples of a certain type, where `type` stands for either positive or negative examples. In our experiments negative examples were substituted by random examples (see Section 2.1 page 4).

```
%used as: compute_Lmodcover(Type,Clause, L-modifiedCoverage)
compute_Lmodcover(Type,(Head:-Body), Cov) :- !,
     '$aleph_global'(atoms_left,atoms_left(Type,Left)),
     compute_Lmodcover(Type, (Head:-Body), Left, 0, Cov).
compute_Lmodcover(Type, ClauseWithoutBody, Cov) :-
     compute_Lmodcover(Type,(ClauseWithoutBody:-true),Cov).

compute_Lmodcover(_, _, [], Cov, Cov).
compute_Lmodcover(Type, Clause, [Inter|Rest], Cov, CovRes) :-
     compute_Lmodcover_interval(Type, Clause, Inter, Cov, Cov1),
     compute_Lmodcover(Type, Clause, Rest,Cov1,CovRes).

compute_Lmodcover_interval(_, _, Start-Finish, Cov, Cov) :-
     Start > Finish, !.
compute_Lmodcover_interval(Type,Clause,Start-Finish,Cov,CovRes) :-
     example(Start, Type, Atom),
%get the L-modified coverage SeqLength for this example:
     parse_exple_for_length(Clause, Atom, SeqLength),
%add L-modified coverage for this example to the total coverage:
```

```
    Cov1 is Cov+SeqLength,
    Start1 is Start+1,
    compute_Lmodcover_interval(Type, Clause, Start1-Finish,
                                             Cov1, CovRes).
```

predicate: parse_exple_for_length/3

This predicate parses an example and returns the length of the sequence if it was parsed sucessfully. If it cannot be parsed, 0 is returned.

```
%used as: parse_exple_for_length(Clause,Example,SequenceLength)
parse_exple_for_length((Head:-Body),Example,SeqLength) :-
% the \+(\+()) are needed to ensure Head and Example
% do not stay unified after the call
    \+((\+((Example=Head, call(Body)))),
    Example=middle(MidSeq,[]),
    length(MidSeq,SeqLength), !.
parse_exple_for_length(_,_,0).
```